Large-Scale Energy-Efficient Graph Traversal: A Path to Efficient Data-Intensive Supercomputing

Nadathur Satish, Changkyu Kim, Jatin Chhugani, and Pradeep Dubey
Parallel Computing Lab, Intel Corporation

Abstract—

Graph traversal is a widely used algorithm in a variety of fields, including social networks, business analytics, and high-performance computing among others. There has been a push for HPC machines to be rated not just in Petaflops, but also in "GigaTEPS" (billions of traversed edges per second), and the Graph500 benchmark has been established for this purpose.

Graph traversal on single nodes has been well studied and optimized on modern CPU architectures. However, current cluster implementations suffer from high latency data communication with large volumes of transfers across nodes, leading to inefficiency in performance and energy consumption. In this work, we show that we can overcome these constraints using a combination of efficient low-overhead data compression techniques to reduce transfer volumes along with latency-hiding techniques. Using an optimized single node graph traversal algorithm [1], our novel cluster optimizations result in over 6.6X performance improvements over state-of-the-art data transfer techniques, and almost an order of magnitude in energy savings.

Our resulting implementation of the Graph500 benchmark achieves 115 GigaTEPS on a 320-node/5120 core Intel® Endeavor cluster with E5-2700 Sandybridge nodes, which matches the second ranked result in the most recent November 2011 Graph500 list [2] with about 5.6X fewer nodes. Our cluster optimizations only have a 1.8X overhead in overall performance from the performance of the optimized single-node implementation, and allows for near-linear scaling with number of nodes. Our algorithm on 1024 nodes on an Intel® Xeon® X5670 Westmere processor (with lower per-node performance) for a large multi-Terabyte graph attained 195 GigaTEPS in performance, proving the high scalability of our algorithm. Our per-node performance is the highest in the top 10 of the Nov 2011 Graph500 list.

I. Introduction

There is a huge challenge in being able to perform "big-data" analysis with the massive growth in data in the past decade through technologies such as cell phones, medical data and social media interactions. Graphs have been used to model many of these interactions, including social network analysis such as Google, Facebook and Twitter; biological systems; ranking web pages and so on [3–5]. Due to the increasing scale of data in these areas, it is important to be able to perform various kinds of analysis on large graphs of Terabytes or more of data [6, 7].

At the scale of data that is now available, graphs cannot be stored on a single compute node, and hence large scale clusters or supercomputers are required. Scaling graph traversal to multi-node clusters has however proved challenging [8]. The importance of solving this problem has led to graph algorithms being proposed as an alternative metric of supercomputer performance [8], and the Graph500 benchmark suite [9] has been proposed for this purpose. The kernel on which Graph500 results are ranked is Breadth-First Search, a representative graph traversal kernel on unweighted graphs. This proceeds in a sequence of steps; each of which iterates over a set of boundary vertices and enqueues their unvisited neighbors to be processed at the next depth. The algorithm outputs the minimum depth and parent for each vertex. This kernel is a basic building block used in graph analysis and has similar characteristics in terms of data access to other graph applications. Graph analysis has also been included as one of the Exascale challenge applications [10, 11].

The total costs for running a cluster are increasing being dominated by electricity costs related to energy consumption [12, 13]. This is projected to get even more expensive in the future. Energy is the product of power and runtime; and one important way to decrease energy consumption is to reduce overall runtime. The overall performance is determined by two factors: the single-node performance and the node-level scaling when running on a cluster. In this work, we start with an optimized single-node graph traversal algorithm [1], and extend it to multiple nodes. We use an ownership approach where each node owns a subset of the input graph vertices (along with the corresponding adjacency data). The boundary vertices at each depth have to be communicated to the nodes that own them. The large volume and high latency involved in this communication is a major performance bottleneck, and is the focus of the optimizations proposed and implemented in this paper.

We first focus on reducing the volume of data communicated. Typical implementations transfer potential boundary vertices and parent for each traversed edge, which takes 16 bytes according to the Graph500 specification [2]. This takes over 96% of the overall runtime. A 2X data volume reduction can be easily obtained by using 4-byte local node ids to encode the original 8-byte values for both vertex and parent ids. There are also many duplicate boundary vertices transferred (with different parents), and we can perform data de-duplication to only transmit unique boundary vertices along with a valid parent. However, these techniques do not yield sufficient compression. Further, the resulting data transfers have high randomness due to the random nature of the vertex and parent id's; and the data transferred has low compressibility [14].

We therefore propose an algorithmic change by decoupling the parent transfers from those of the boundary vertices. Instead of pre-emptively sending potential parents, we only transfer parents when a vertex is first visited. Since this

SC12, November 10-16, 2012, Salt Lake City, Utah, USA
978-1-4673-0806-9/12/$31.00 ©2012 IEEE

happens only once per vertex, this dramatically reduces overall data volumes by 1.8-1.9X. Additionally, the boundary vertices that are transferred can be now rearranged in a way that entropy is reduced [15, 16]. We use a bit-vector to encode these rearranged vertices to achieve an overall compression of 6.2-9X as compared to sending 8 bytes per edge.

In order to reduce the performance impact of the latency of data communication, we perform two optimizations. First, we overlap the communication with the computation done on each node. The intent is to keep the nodes busy while communication occurs. Second, since the communications of the parents typically involves small message transfers with high latency impact, we try to combine these transfers with boundary vertex communication. However, there is a dependency between these communications at a given depth; parent communications can only start after all boundary vertices have been received and processed. Therefore, we pipeline the communication by combining the boundary vertex transfer at a given depth with parent transfers for preceding depths (applying the concept of software pipelining). These two techniques lead to an overall performance improvement of 2.1X.

The combination of our compression and latency hiding techniques results in over 6X per-core performance over recently published cluster implementations of graph traversal systems [17, 18]. We obtain 6.6X performance improvements over our baseline performance using 8 bytes/edge. Reducing the volume of data transfer also reduces the expended power. This power reduction along with performance improvements results in about 8.1X improvement in energy consumption over our baseline cluster implementation. To the best of our knowledge, this is the first paper that measures and reports energy consumption for Graph500. Our resulting implementation of the Graph500 benchmark achieves 115 GigaTEPS on a 320-node/5120 core Intel Endeavor cluster with E5-2700 Sandybridge nodes, which matches the second ranked result in the most recent November 2011 Graph500 list [2] with about 5.6X fewer nodes. In terms of energy consumption, we use the recent Intel NodeManager [19] API [19] for measuring energy, and achieve energy efficiency of 1.5 MTEPS/Watt – this can serve as a baseline for future comparisons. Our cluster optimizations only have a 1.8X overhead in overall performance from the performance of the optimized single-node implementation, and allows for near-linear scaling with number of nodes. Our per-node performance is the highest in the top 10 of the Nov 2011 Graph500 list[1]. We also ran our algorithm on 1024 nodes of Intel Westmere and Sandybridge clusters for a large scale graph and obtain performance of 195 and 270 GigaTEPS on the two platforms showing high scalability. Our compression and latency hiding techniques developed in this paper are applicable to a wide class of graph traversal algorithms such as connected components [20], shortest paths [21] and betweenness centrality [22].

[1]Our previous submission (ranked #6) to the Nov. 2011 Graph500 list [2] based on some of the algorithmic techniques described here already achieved the highest per-node performance in the top 10; per-node performance in this paper is about 2X higher due to algorithmic and machine differences.

II. Problem Definition

The graph problem is a breadth-first search on an unweighted graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. The adjacency array is denoted by $\mathcal{A}dj$. Given a starting vertex s, the algorithm produces (as output) for each vertex $v \in \mathcal{V}$, (a) the shortest distance in terms of number of edges from s to v, and (b) the parent of v; defined as the previous vertex in the shortest path from s to v.

A naïve BFS algorithm proceeds in *steps*, wherein each step iterates over the set of vertices in the boundary set $\mathcal{BV}^C$, looks up the $\mathcal{A}dj$ structure for each vertex and updates the depth (*Depth*) (and parent (*Parent*)) of unvisited neighboring vertices and enqueues them to the set of boundary states for the next step ($\mathcal{BV}^N$). There is a synchronization point between steps. The challenge for building an efficient multi-node BFS traversal is to (1) reduce the impact of random accesses due to the spatially incoherent accesses to $\mathcal{A}dj$, *Depth*, *Parent*, etc. on each node. (2) provide an efficient de-centralized mechanism to figure out whether a neighboring vertex has been visited or not. This should incur *low overheads* in terms of latency and and bandwidth consumed. (3) Scale with increasing number of nodes to achieve a high performance and energy efficiency.

Graph500 benchmark Specification: The Graph500 benchmark specifies additional requirements. Graph $\mathcal{G}$ is specified to be a scale-free undirected Kronecker graph. Each edge must be stored using 8 bytes, and each vertex stored with at least 48 bits. The number of undirected edges is 16 times the number of vertices. Performance is reported in TEPS (traversed edges per second), which is the ratio of number of undirected edges traversed to the time taken to complete BFS. The BFS code must be run with 64 different starting vertices to obtain statistics on runtime.

We convert the undirected graph to a directed graph by creating two copies of each undirected edge, one for each direction, doubling degree ρ to 32. Since we traverse twice as many edges as the number of edges in the Graph500 input, we divide our performance results by 2 when reporting our performance results in TEPS (to match Graph500 specifications).

III. Algorithm

In this and future sections, we use the notation described in Table I to describe the optimized multi-node BFS algorithm. Our multi-node BFS traversal uses an optimized single-node BFS traversal [1] algorithm as the building block for extension to multiple nodes, while optimizing for MPI traffic and reducing the computational and transfer overhead on each node to get an effective single-node efficiency as close as possible to the reported single-node performance. We briefly describe the single-node algorithm below for ease of explanation of our scheme (refer to [1] for details).

A. Single-Node BFS Traversal [1]:

The algorithm maintains a cache-resident auxiliary bit structure (denoted by *VIS*) to check whether a vertex has already been assigned a depth. *VIS* is divided into $\mathcal{N}_{VIS}$ parts such that

Notation	Description	Notation	Description
$\|\mathcal{V}\|$	Number of Vertices in the graph.	$\|\mathcal{E}\|$	Number of Edges in the graph.
$\|\mathcal{V}'\|$	Number of Vertices assigned a depth during the BFS traversal.	$\|\mathcal{E}'\|$	Number of Traversed Edges during the BFS traversal.
$\|\mathcal{V}^s\|$	Number of vertices on node s.	$\|\mathcal{E}^s\|$	Number of edges on node s.
ρ'	Average degree of the vertices assigned a depth ($=\mathcal{E}'/\mathcal{V}'$).	$\mathcal{D}$	Depth of the Graph.
$\mathcal{M}$	Number of computational nodes.	$\mathcal{P}$	Number of cores on each node.
$\mathcal{B}_M$	Achievable Main Memory B/W (in GB/sec) on each node.	$\mathcal{F}req$	Core Frequency (in GHz).
$\|VIS\|$	Size of *VIS* Array (in Bytes). Equals $\|\mathcal{V}\|/8$ Bytes.	$\mathcal{L}$	Cache-Line Size (in Bytes).

TABLE I: **Notation used in the paper.**

```
// Compute |V_NS| and N_VIS.
// Allocate Adj, Depth, Parent, VIS, BV_t^C , BV_t^N , PBV_t , and Initialize
for (step=1; ; step++)
    Reset BV_t^N
    Divide_BV_t^C_among_threads( BV_t^C )
    for all vertex u ε assigned BV_t^C
Phase-I         Perform_Prefetches for Adj   /* Prefetch neighbor list */
                Perform_SIMD_Binning_PBV_t ( u, Adj, PBV_t ); /* Partition neighbor
                                                among N_VIS bins */
    endfor
    barrier();

    Divide_PBV_t_among_threads ( PBV_t )
    for all vertex v ε assigned PBVt
Phase-II        u = Parent ( v);
                UPDATE_BV_VIS ( u, v, step );
    endfor
    Rearrange (BV_t^N)
    barrier();

    new_num_boundary_verticies = Sum (BV_t^N)
    if (new_num_boundary_verticies == 0) break;
    SWAP(BV_t^N , BV_t^C);
endfor
```

Fig. 1: **Single-Node BFS Graph Traversal [1].**

each fits entirely in LLC and is also updated in an atomic-free fashion to reduce the impact of latency and overhead on performance. The algorithm iterates in **steps**, with each step consisting of the following *two* phases (Fig. 1):

In Phase-I, divide the set of boundary states ($\mathcal{BV}_t^C$) evenly between threads. Each thread is thus assigned a list of vertices for which $\mathcal{A}dj$ needs to be accessed. However, since successive vertexes may point to spatially non-contiguous regions of memory, this access pattern cannot be captured by the hardware pre-fetcher. Hence, while accessing $\mathcal{A}dj$ for a vertex, issue pre-fetch instructions for the $\mathcal{A}dj$ of a few subsequent vertices into L1 cache. For each accessed neighbor, compute its bin_id (based on the vertex_id) and append it to the relevant partition ($\mathcal{PBV}_t$). In order to facilitate neighbor computation, pair of (vertex_id, parent_id) is enqueued. As mentioned above, $\mathcal{N}_{VIS}$ partitions are created for cache-friendly *VIS* accesses. There is a barrier at the end of Phase-I.

In Phase-II, each partition is evenly divided amongst the threads. While operating on a given partition, for each vertex, execute UPDATE_$\mathcal{BV}$_$\mathcal{VIS}$ (atomic-free update to the *VIS* array), which appends the list of vertices for which *VIS* was updated into the list of boundary states for the next step ($\mathcal{BV}_t^N$). In case the vertex is not visited, it *depth* and *parent* are also updated. The algorithm terminates when no vertices are added to $\mathcal{BV}_t^N$.

The above single-node algorithm achieves a performance of around 684 MTEPS on our 2.6 GHz dual-socket SandyBridge system for a scale=26 graph, which closely matches their reported performance after accounting for the increased memory bandwidth on our system as compared to theirs.

B. Scalable Multi-Node Algorithm:

One major reason why graph traversal is difficult to scale is that traversal requires large inter-node communication volumes, and inter-node bandwidth is limited. For example, it takes about 20.8 cycles to transfer an edge of 8 bytes over a 1 GBps InfiniBand link (assuming node frequency of 2.6 GHz) – this is over 10× worse than the single node performance. Another reason is the high latency of inter-node communication. The latency of inter-node transfers of data structures like the $\mathcal{PBV}$ data is orders of magnitude higher than shared memory access. As a result of these factors, **additional optimizations are required** for multi-node implementations.

In this work, we provide a scalable solution to graph traversal using a combination of techniques discussed below. We obtain the most throughput efficient multi-node implementation in the Graph500 list for Terabyte scale graphs or higher.

Data division between nodes: The $\mathcal{A}dj$ array is distributed between nodes. We distribute vertices amongst the nodes, with each node holding the Adjacency data for the vertices it owns. Since there can be a large variance in the degree of the vertices, we divide vertices such that the size of Adjacency structure on a node is similar to the other nodes. Each node maintains the the range of vertices owned by every other node. For a node m, we denote the starting vertex as $\mathcal{SV}^m$, and hence the range of vertices owned by node m equals [$\mathcal{SV}^m$ .. $\mathcal{SV}^m$ + $|\mathcal{V}^m|$). Each node is responsible for traversing the subset of $\mathcal{BV}$ vertices that it owns at each depth. The *VIS* data structure is also divided among nodes in the same way as $\mathcal{A}dj$, and nodes are responsible for updating the *VIS* as well as output depth and parent arrays for the vertices they own. These data structures are stored in a NUMA aware fashion on each node as in the single-node algorithm.

Algorithm Sketch: Figure 2 shows the complete algorithm for multi-node traversal. During the traversal of a particular depth, each node (m) looks up neighbors for its subset of the $\mathcal{BV}$ list in a similar fashion as the single-node algorithm; and generates a $\mathcal{PBV}$ array. The vertices in $\mathcal{PBV}$, however, may belong to any node. This then requires the vertices in $\mathcal{PBV}$ to be **communicated** to the nodes that own this data. Each node receives and collects the $\mathcal{PBV}$ entries that belong to it from all other nodes, and then continues with updating the $\mathcal{DP}$ arrays and generating the $\mathcal{BV}$ array for the next depth (as in the single node implementation).

For efficient communication of the $\mathcal{PBV}$ vertices from a node to others, we send a single message with all entries to be communicated. This requires us to **bin the $\mathcal{PBV}$ entries of each node into $\mathcal{M}$ bins** ($\mathcal{M}$ being the number of nodes), with each bin ($\mathcal{PBV}_m$) obtaining all $\mathcal{PBV}$ entries owned by node

m. This binning can easily be combined with the binning into $\mathcal{N}_{VIS}$ bins done for maintaining *VIS* in cache – this requires us to bin into $\mathcal{M} \cdot \mathcal{N}_{VIS}$ bins in total. For small to medium scale clusters of up to 2000 nodes, we find that this additional binning does not result in significant performance degradation, with a low latency impact due to TLB misses.

Due to the limited node-to-node bandwidth, the major performance component is the amount of data transferred between nodes. While traversing $\mathcal{A}dj$ for the vertices that a node owns, it appends pair of (vertex_id, parent_id) to $\mathcal{PBV}_m$. After traversing through each step, this data is then transferred to the relevant nodes. We now focus on this data communication and *propose algorithmic changes to reduce the amount of data transferred*, and hence improve the performance with *minimal overheads on the computation side*. It is critical to keep the overheads low since these reductions are applied during the traversal algorithm, and hence the costs are added to the BFS traversal cost.

C. Node-To-Node Traffic Reduction through Compression:

Scheme 1. 4-byte each (vertex_id, parent_id) pairs: Each vertex is stored using a 64-bit id, and hence a naïve implementation would send 8-bytes for both vertex_id and parent_id, for a total of 16-bytes per edge.

However, given the current main memory sizes on each node, $|\mathcal{V}^m|$ (the number of vertices on a node m) is typically less than 2^{32}, which would require > 1 Terabyte of memory on each node. Since each node maintains the range of vertices on all the nodes, it can transfer the local_vertex_id (obtained by subtracting the minimum vertex_id owned by the node from the vertex_id), and correspondingly its local_parent_id to each node. The total size reduces to **8-bytes transferred per edge**, and this serves as the *base-case* for all our further improvements and optimization's.

Scheme 2. Data De-duplication: We now focus on the number of $\mathcal{PBV}$ pairs that are transferred between nodes. It is easy to see that the total number of pairs in the $\mathcal{PBV}$ arrays transmitted between all pairs of nodes and all depths combined is nearly the number of traversed edges in the graph (E'). However, the total number of unique entries in these $\mathcal{PBV}$ arrays can only be the number of visited graph vertices, a factor of ρ' (normalized graph degree) times smaller than the number of edges. This means that each entry in $\mathcal{PBV}$ is duplicated around ρ' times. Also, since the resultant output is a tree, each vertex indeed only needs to store one parent, and hence there are a lot of redundant pairs being transferred due to the underlying distributed non-sharing multi-node setup.

While it may not be possible to eliminate all these duplicates since they may be present in different nodes and at different depths, at least the **duplicates within each node at each depth can be eliminated**. For Graph500 Kronecker graphs, $\rho' \sim 64$, and most of the edges are traversed in a *few depths*, which increases the likelihood of duplicates.

Hence, before transferring $\mathcal{PBV}$ pairs, all pairs with the same local_vertex_id are combined into a single pair with the same local_vertex_id, and arbitrarily chosen local_parent_id from various such pairs. We achieve this using an *efficient low-overhead* bit-structure scheme, explained in detail in Sec. IV. The achieved reduction in node-to-node traffic is a function of graph characteristics, and increases with increasing average degree of the vertices. In practice, for Graph500 Kronecker graphs, we achieved a reduction varying between 1.2–1.4× (details in Sec. V) for various graph sizes.

Scheme 3. Variable Length Compression: As a limit study, we performed on-the-fly compression of the data transferred between nodes. Specifically, we used bzip2 [14] compression and decompression algorithms for our purpose. For this limit study, we also sorted the pairs [15] on the local_vertex_id to reduce the resultant entropy, and increase the amount of compression of the data being transferred. However, the achieved reduction in transferred data varied between 1.05–1.15×, while incurring large overheads (around 11-23× increase in computation time).

We therefore propose the following algorithmic change in the content being transferred in $\mathcal{PBV}$, as described next.

D. Separate Parent_Ids from Vertex_Ids

As noted earlier, for each vertex that is assigned a depth, only one parent needs to be stored. However, we pre-emptively send the parent_id along with the vertex_id, even though only one potential parent_id needs to be transferred. Hence we change this protocol, and instead each node only sends the relevant vertex_id as part of the $\mathcal{PBV}_m$. Once the node (say m) receives all the $\mathcal{PBV}_m$ packets from the different nodes, it checks the local *VIS* array, and in case the vertex has not been assigned a depth so far, it appends that vertex to a **Parent_Query** ($\mathcal{PQ}$) packet that is sent back to the original node from which $\mathcal{PBV}_m$ was received. Note that for any given vertex that is updated during a step, the parent request is **only sent to one of the source nodes**, not to all the nodes from which it was received.

After traversing through all the vertices, these Parent_Query packets are then transferred back, and the source nodes compute the relevant parents and send back **Parent_Response** ($\mathcal{PR}$) packet to the destination node (m), which is then used to set the parent. This leads to a dramatic reduction in node-to-node traffic as the number of parent related transfers for each vertex is now **reduced to exactly two, as opposed to average of** ρ in the other implementations. This has a potential to reduce the traffic by $(\frac{2\rho}{2+\rho})$ ×.

An efficient low-overhead parent computation algorithm is described later in Sec. IV. As far as the reduction is concerned, we achieve an additional 1.8–1.9× reduction in node-to-node traffic. The other advantage of sending separate vertex_ids separately is that they can now be reordered in any fashion – since they all represent the set of vertices at a particular depth. We can hence sort them to compress them to further reduce the traffic, as described next.

Scheme 4. Bit-Vector Representation: For a given $\mathcal{PBV}_m$ packet with a list of vertices, we also maintain a bit-array ($\mathcal{B}it\mathcal{A}rray_m$), with 1-bit per vertex that stores a 1 for a vertex that is contained in $\mathcal{PBV}_m$ and 0 otherwise. For a

```
// Divide data structures into per-node structures
// Allocate Adj, Depth, Parent, VIS, BV^C , BV^N , and Initialize
for (step=1; ; step++)
        Reset BV^N
        PBV_[1..M] = Phase-I (BV^C , Adj)
        barrier();
        for each node m
                CPBV_m = Compress (PBV_m)
                Send CPBV_m to node m
        endfor
        for each node n
                Receive CPBV'_n from node n
                PBV'_n = Decompress (CPBV'_n)
                BV^N = Phase-II (PBV'_n) U BV^N    /* Update BV^N , DP, VIS */
        endfor
        for each updated vertex v in BV^N
                Update_Parent (v, Parent)  /* Parent queried from the node that
                                              has sent vertex v to this node */
        endfor
        new_num_boundary_states = MPI_Allreduce ( BV^N )
        if (new_num_boundary_states == 0) break;
        SWAP(BV^N , BV^C);
endfor
```

Fig. 2: **Multi-Node BFS Graph Traversal Algorithm.**

node m, the total size of ($\mathcal{B}it\mathcal{A}rray_m$) equals $8\lceil\frac{|\mathcal{V}^m|}{8}\rceil$ bytes. In case the size of $\mathcal{B}it\mathcal{A}rray_m$ is less than size of $\mathcal{PBV}_m$, we transfer $\mathcal{B}it\mathcal{A}rray_m$, which represents the same set as $\mathcal{PBV}_m$, except that the vertices are now represented in a sorted fashion, which does not change the results for BFS traversal. Note that sending $\mathcal{B}it\mathcal{A}rray_m$ automatically performs data de-duplication as each vertex is only sent once. For depths with large traffic, this technique leads to around **0.9–1.3 bytes transferred per edge**, depending on the number of nodes $\mathcal{M}$.

Although there are further compression possibilities (for instance delta encoding and variable length compression), the ones mentioned above suffice for the up to a few thousand nodes that we tested our implementation on. Note that all these data reduction/compression techniques need to be applied at run-time and also the data needs to be decompressed and all the times need to be taken into account for the final run-time. Hence there is need to strike the right balance between expensive compression techniques and the resultant pay-off in terms of memory traffic reduction.

IV. Implementation Details

We use an Intel® Xeon® E5-2670 Sandybridge based cluster, using pthreads to exploit thread-level parallelism and SSE/AVX [23] to exploit data-level parallelism. We use Message Passing Interface [24] (MPI) (we use the Intel® MPI [25] library 4.0.3) calls for inter-node communication.

A. Implementation Steps:

We now describe the details of the various steps outlined in Figure 2. For each depth, the algorithm performs the following: **Step 1: Traverse $\mathcal{A}dj$ and fill $\mathcal{PBV}$:** At each depth, the threads in the node divide this $\mathcal{BV}$ among themselves, and perform a NUMA-aware cache friendly traversal of the $\mathcal{A}dj$ data structure as in **Phase-I** of the single node algorithm. After this traversal, the neighbors are now written out to a $\mathcal{PBV}$ array that is binned into $\mathcal{M}\cdot\mathcal{N}_{VIS}$ bins as described in Section III. To reduce the instruction overheads, we compute the bin id's of 8 simultaneous neighbors using AVX instructions. The writes into $\mathcal{PBV}$ are however done using scalar code since they involve gathers/scatters. This phase is bound by the memory bandwidth required to perform the reads and writes of the neighbors. We perform 2-phase partitioning if the number of bins is greater than 1024 to reduce the impact of TLB misses. For the remainder of this paper, for convenience of explanation, we assume there are only $\mathcal{M}$ bins; each bin logically holds the concatenation of the $\mathcal{PBV}$ entries in the $\mathcal{N}_{VIS}$ bins belonging to a node. At the end of the partitioning, the threads within the node execute an intra-node barrier.

Step 2: Compress PBV and Send: In this step, the binned $\mathcal{PBV}_m$ entries in each bin $m \in 1..\mathcal{M}$ are sent to destination node m. To reduce the data traffic between nodes, we use Scheme 4 as described in Section III. We use bit-arrays $\mathcal{B}it\mathcal{A}rray_m$ for each destination node m to (1) perform de-duplication of the entries of $\mathcal{PBV}_m$; and (2) to encode $\mathcal{PBV}_m$ using a small number of bits. The $\mathcal{B}it\mathcal{A}rray_m$ array has only one bit per vertex belonging to node m, equal to $|\mathcal{V}^m|$ bits. To populate $\mathcal{B}it\mathcal{A}rray_m$, we **divide the destination nodes evenly among the threads**. Each thread then populates the $\mathcal{B}it\mathcal{A}rray_m$ array for one or more nodes m as follows: (1) Scan the $\mathcal{PBV}_m$ array corresponding to node m. (2) For each entry k in $\mathcal{PBV}_m$, set bit $(k-\mathcal{SV}_m)$ in $\mathcal{B}it\mathcal{A}rray_m$ to 1, where $\mathcal{SV}_m$ is the first vertex belonging to node m. We perform the index computations using AVX operations, but perform bit updates using scalar operations. This $\mathcal{B}it\mathcal{A}rray_m$ array is written out to a compressed $\mathcal{CPBV}_m$ array and sent to node m.

To calculate the total inter-node communication, each node receives $|\mathcal{V}^m|$ bits from every other node. Hence the global data transfer is about $|\mathcal{V}^m|\times\mathcal{M}^2 \sim |\mathcal{V}|\times\mathcal{M}/8$ bytes. For $|\mathcal{V}| = 2^{33}$, $\mathcal{M} = 320$ nodes and assuming graph degree $\rho = 32$, this is about 1.25 bytes/edge. This data transfer will occur at each depth, irrespective of the size of the $\mathcal{PBV}$ arrays and the number of duplicates among $\mathcal{PBV}$ entries. Thus, if the number of de-duplicated edges is small, this scheme is inefficient, and we fall back to sending 4 bytes per de-duplicated edge. In practice, most edges are traversed at two depths; consequently we use the bit-array scheme at those two depths and use 4 bytes to transfer the rest of the edges. After we account for de-duplication, most depths except the two depths where most edges are traversed have negligible impact on the amount of data communicated; we communicate 1.25 bytes/edge/depth *2 depths = **2.5 bytes/edge of inter-node communication**.

In terms of the computation performed, the operations performed are minimal, and performance is limited by the memory bandwidth requirements. We found that the **overall overhead of compression is about 1.4X**.

Step 3: Receive and Decompress PBV: The compressed $\mathcal{CPBV}_m$ arrays from each source node m are now decompressed, and the $\mathcal{PBV}_m{}'$ arrays are written out. As opposed to step 2, as each $\mathcal{CPBV}_m$ array arrives, we have all the threads *co-operatively* process the array. This is because inter-node latencies are variable; and different messages may arrive out-of-order at any time; hence it is most efficient to *process each*

packet as they are received. Each thread is responsible for a fixed range of vertex id's that is equally partitioned among the vertices belonging to the node. As an array $\mathcal{CPBV}_m$ is received from node m, each thread looks at its section of the array and for each bit k set to 1, writes out a 4-byte value to a per-thread $\mathcal{PBV}_m{}'$ array equal to $k+\mathcal{SV}_{node_id}$, where $\mathcal{SV}_{node_id}$ is the starting vertex of the node performing the decompression. We use SIMD operations to read in and check 256 vertices at a time. The performance is mainly limited by the memory bandwidth of reading the $\mathcal{CPBV}$ and writing the $\mathcal{PBV}_m{}'$ arrays. The overall **overhead of uncompressing the data is about 1.3X**. Together with the compression overheads of about 1.4X, Steps 2 and 3 together have a relatively **low overhead of 1.7-1.9X** to the single node computation costs, however **reducing MPI transfers by over 3X**.

Step 4: Check *VIS*: After each decompressed $\mathcal{PBV}'$ array is written out, it is checked against the per-node *VIS* data structure. Note that vertices in all of the $\mathcal{PBV}'$ arrays are guaranteed to belong to the node processing them. Each thread looks at its $\mathcal{PBV}_m{}'$ array, and performs the efficient UPDATE_$\mathcal{BV}$_$\mathcal{VIS}$ step as in **Phase-II** of the single node algorithm. The depth of each unvisited vertex $v \in \mathcal{PBV}'$ is updated to the current depth; and v is added to the $\mathcal{BV}^N$ array for the next depth.

Step 5: Fill in Parent Queries and Transfer: For each vertex whose depth is updated in Step 4, the corresponding parent vertex is identified as follows: As the depth of any vertex, say v, is updated in Step 4, we first identify the node from which this vertex originated. Let us say it came from node m. We then enqueue the local offset 4-byte value $v-\mathcal{SV}_{node_id}$ into an Parent Query array for node m (denoted as $\mathcal{PQ}_m$). The idea is that node m, having sent vertex v, is responsible for identifying the parent of v (which should be one of the vertices belonging to node m). To avoid locking, we perform a division of nodes among threads, and each thread only enqueues vertices into a subset of $\mathcal{PQ}_m$ arrays. After all $\mathcal{CPBV}$ arrays are received, each $\mathcal{PQ}_m$ array is sent to node m.

Step 6: Process Parent Queries, Send Parent Replies:

Consider a node (n). For each node (m), it maintains a list of (vertex_id, parent_id) pairs (sorted by the vertex_id) for each of the vertices that were the neighbors for some vertex owned by n. We refer to this structure as Reverse_Adj^m, for the node m. Node n maintains $\mathcal{M}$ such structures – one for each node.

Consider Fig. 3. It receives a set of $\mathcal{PQ}_m$ from each source m, and must identify the parents of each vertex in $\mathcal{PQ}_m$. Consider a vertex v in $\mathcal{PQ}_m$. Since Reverse_Adj^m is already sorted by vertex_id, one can find the parent using a binary search, and in case of more than one match, return the parent with the appropriate depth. However, this is computationally expensive [27] and adds substantially to the overhead.

Instead, we maintain an auxiliary structure, a hash table, that performs an expected O(1) number of comparisons to find the first tuple in Reverse_Adj^m with v. We set the size

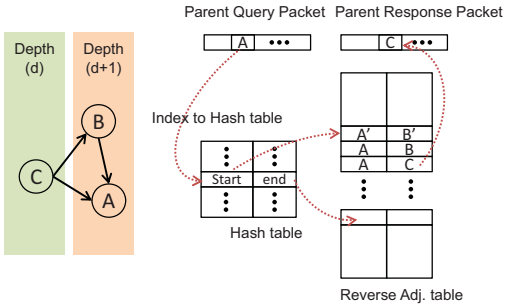


Fig. 3: **Processing parent queries through a lookup of a Reverse Adjacency structure storing (vertex_id, parent_id) tuples. An auxiliary hash table is used to perform O(1) lookups. The hash table stores a range of entries in the Reverse Adjacency to traverse. We return the parent_id of the first (vertex_id, parent_id) tuple in this range such that (1) vertex_id matches the query; and (2) the parent_id is at depth (D-1). We perform this search using a SIMD algorithm [26]. For the query vertex A, tuples (A',B') and (A,B) do not meet these conditions. The parent returned is C.**

of the hash table to be around |Reverse_Adj^m|/16, and choose a hash function belonging to the class of strongly 2-universal functions [28]. The overhead of storing the hash table is <2% of the overall space occupied. This technique reduces the overhead of parent computation by more than an order of magnitude as compared to binary search for large graph sizes.

Since the communication and computation for Steps 5 and 6 are only performed once per vertex, the cost of these steps is small. For instance, the MPI traffic is only about 0.125 bytes/element, about 5% of overall traffic. With our efficient parent query scheme, the computation costs for Steps 5 and 6 add about 1.15X to overall computation time.

Step 7: Update Parent: The Parent responses obtained in Step 6 are used to update the Parent arrays.

Step 8: Termination Condition Check: The algorithm at the end of each depth performs a `MPI_AllReduce` call whereby it adds up the number of entries in the $\mathcal{BV}$ arrays across all nodes. If this sum is 0, all nodes decide to exit the loop. However, this call results in significant latency; we eliminate this call as shown in the last part of this section.

B. Overlap computation and communication:

Our baseline implementation sends out three types of packets between each pair of nodes, one for $\mathcal{CPBV}$, one for $\mathcal{PQ}$ and one for $\mathcal{PR}$. Due to limited inter-node communication bandwidth, the transfer of these packets can take up a major portion of the overall runtime. During this time, the cores are idle. Further, there may be network latency effects causing certain packets to arrive later than others, leading to long idle times. Finally, load imbalance where one node traverses more neighbors at a depth and sends out more data than other nodes can also cause variations in the time at which packets are sent by different nodes, again resulting in core idling.

In order to increase overall efficiency, we can overlap the communication with computation on the cores. There are significant opportunities to do this. Consider Figure 4(a). Since each node receives $\mathcal{M}$ packets for each packet type, as soon as a packet arrives, it can be processed by the

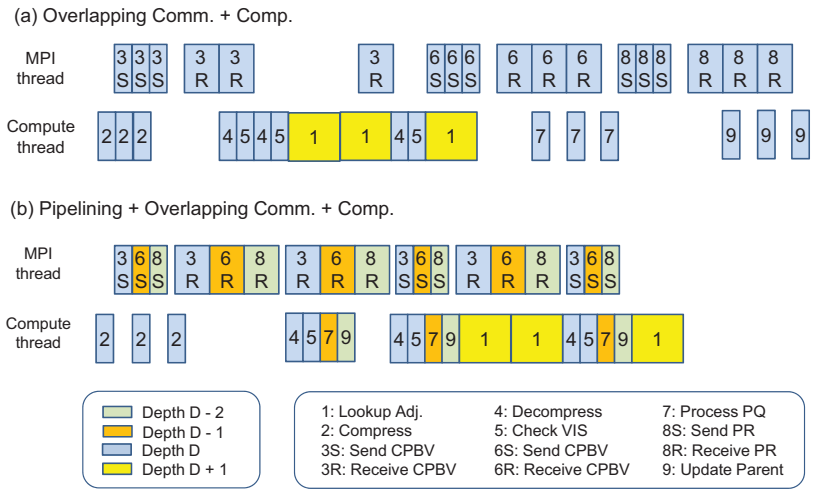


Fig. 4: **Latency optimizations for graph traversal. Various algorithmic steps for a single BFS depth in the steady state are shown. Our optimizations include (a) Overlapping communication and computation and (b) Pipelining. Processing of received $\mathcal{CPBV}$, $\mathcal{PQ}$ and $\mathcal{PR}$ packets from some nodes can occur in parallel with waiting for packets from other nodes (which occasionally incur long latency). Three packets of each type are shown in the figure. Additionally, Step 1 (Lookup Adjacency) is pre-emptively done for depth (D+1) during long communication wait times. Data transfers as well as the processing of $\mathcal{CPBV}$ arrays at depth (D) can be pipelined with those of $\mathcal{PQ}$ at depth (D-1) and $\mathcal{PR}$ at depth (D-2). This leads to larger message transfers and improved latency. The colors indicate the depths for which different implementation steps are performed. Overlapping results in 1.6X performance improvement over the baseline of performing steps 1–9 sequentially. Pipelining further improves performance by 1.3X, leading to an overall 2.1X performance increase over the baseline.**

node. During this processing time, other packets can continue transmission. This can be done for each of the packet types, shown in Figure 4(a). Similarly, the sends of $\mathcal{CPBV}$ packets can occur in parallel with generating the remaining packets. In practice, **overlapping** yields benefits of about **1.5-1.7X** in overall performance.

In order to implement this, we use a "hybrid" threading approach. We dedicate one thread (hereafter referred to as MPI thread) to handle all operations associated with MPI including system calls and copying data from user to MPI internal buffers and back. All other threads perform computation tasks. We use asynchronous MPI calls that allow for non-blocking communication. At each depth, the MPI thread first posts a set of receive requests using `MPI_Irecv` calls. These calls notify the MPI communication stack to be ready to receives a set of messages (containing $\mathcal{CPBV}$ entries), but do not block. Once these receives are issued, the MPI thread blocks waiting on at least one message to become ready to send. When at least one $\mathcal{CPBV}$ packet is generated by a compute thread in Step 2, the thread generating the packet signals the MPI thread using a set of shared variables (one per packet) to indicate that the send can begin. The MPI thread, which was polling on this set of variables, issues a `MPI_Isend` call to initiate the sending of the $\mathcal{CPBV}$ packet (and resets the variable). When at least one packet has been sent, the MPI thread then uses the `MPI_Testany` call, which takes in a set of receive requests posted earlier, and checks if any of them have been received. This is a non-blocking call. In case a packet has arrived, then the MPI thread puts the $\mathcal{CPBV}$ entries into a user-space array (visible and coherent across all threads), uses a set of shared variables (one per message per compute thread) to notify the other threads that a message has been received. It then goes back to the polling on whether any other messages are ready to send. We do the `MPI_Testany` call as an optimization to allow maximum overlap of sends and receives. When all threads finish Step 2 and all messages have been sent, the MPI thread checks if any receives are yet to happen in which case it issues a blocking `MPI_Waitany` call which blocks until at least one packet arrives. The MPI thread then notifies compute threads that the packet has arrived, and then goes back to blocking for remaining packets.

On the computation side, once the compute threads finish Step 2, they are blocked waiting for the MPI thread to signal that at least one message has arrived. Once a message arrives, the threads then work cooperatively to decompress and process the message. Once all $\mathcal{CPBV}$ messages have arrived and the $\mathcal{PQ}$ packets generated, the MPI thread is signalled to start communication of the $\mathcal{PQ}$ packets – again in overlapped fashion with the processing of these packets. A similar step occurs for the $\mathcal{PR}$ packets.

Overlap of Step 1 with communication: While the above helps hide some of the communication latency, the problem of load imbalance remains. We found that much of the load imbalance came from Step 1, where different nodes could traverse very different number of edges at specific depths. This can occur in spite of our even static division of $\mathcal{Adj}$ among the nodes; due to dynamic variations for specific depths. This load imbalance is exposed as long MPI wait times for the nodes that finish Step 1 early but cannot make progress waiting for $\mathcal{CPBV}$ packets from nodes that only complete Step 1 later. Since most packets have already arrived and been processed, there is little computation that can be done until these few straggling packets arrive. In practice, we found this load imbalance be as high as **20 - 30%** of overall runtime.

In order to better utilize this wait time (when there is no work that can be done by the node), we *pre-emptively perform Step 1 of the next depth*, using the $\mathcal{BV}'$ array computed so far (shown in Figure 4(a)). When the remaining packets arrive, they are then processed and Step 1 is done for those packets as well. However, there are a few important considerations: (1) we need to ensure that Step 2 is completed by all threads before Step 1 for the next depth can commence; this is to avoid write-after-read conflicts. We ensure this by having each thread in a node set a bit in a signal array when they finish Step 2; each thread only starts Step 1 after ensuring all bits are set to 1. (2) ideally, Step 1 should only be done when no other useful work can be done. In practice, this is hard to determine. As a trade-off, we wait for a certain number of packets (equal to 10% of $\mathcal{M}$) to arrive before we start Step 1 for the next depth. This allows for sufficient opportunity to launch Step 1 while not interrupting Steps 3, 4 and 5.

C. Pipelining communication steps:

While we can utilize the processor during idle network message wait times by overlapping computation with communication, very long network latencies will still have significant impacts on performance. This is especially true for small messages, namely those of the $\mathcal{PQ}$ and $\mathcal{PR}$ arrays, for which network bandwidth utilization is low. Another source of latency is the `MPI_Allreduce` call for termination checks. To overcome these problems, we combine the $\mathcal{PBV}$, $\mathcal{PQ}$, $\mathcal{PR}$ and the boundary count for termination checks into one single message. Thus we send **one large message per depth** rather than multiple smaller messages.

However, there are dependencies between the receipt and processing of the $\mathcal{PBV}$ packets for a given depth (d) in Step 3 and the generation of the $\mathcal{PQ}$ packets for the same depth in Step 5. A similar dependency also exists between processing $\mathcal{PQ}$ packets and generation of $\mathcal{PR}$ packets at a given depth (d). Therefore, combining message types at the same depth is not possible. Consequently, we attempt to **combine message types at different depths** using pipelining.

Consider Figure 4(b). The figure shows the steady state where $\mathcal{PBV}$ packets at depth (d) are combined with the $\mathcal{PQ}$ packets for depth (d-1) and the $\mathcal{PR}$ packets at depth (d-2). There is also a prolog where only $\mathcal{PBV}$ arrays for depth 0 are sent, and then $\mathcal{PBV}$ for depth 1 and $\mathcal{PQ}$ arrays for depth 0 are sent. There are two final epilog steps where $\mathcal{PQ}$ and $\mathcal{PR}$ arrays for the last depths are sent. The prolog and epilog are not shown in the figure. The computation steps are similarly pipelined, where Steps 3, 4 and 5 process packets at the current depth d, Step 6 processes packets at depth (d-1) and Step 7 processes packets at depth (d-2).

D. Reverse Traversal scheme:

It has been previously observed that for Graph500 inputs, most edges are traversed at one or two depths [1, 29]. Moreover, these depth(s) occur after most vertices that will eventually be assigned a depth are already traversed. When such a depth is reached, there are a large number of vertices in the $\mathcal{PBV}$ arrays, but most of them have already been visited. It is then beneficial to switch to a *reverse traversal*, where we look at all remaining unvisited vertices and attempt to find their depths and parents. Since one of the neighbors of each unvisited node has to be the parent, this only requires checking the depth of each neighbor of an unvisited vertex until a visited neighbor is found. This can potentially avoid traversing a large number of edges and has been shown to achieve high performance on single-node systems [29]. In a multi-node setting, we query the nodes that own the neighbors of unvisited vertices to check the depths; this adds some latency to our approach. In practice, incorporating the reverse traversal scheme into our optimized implementation results in overall benefits of around 1.4X in our final implementation.

V. Evaluation

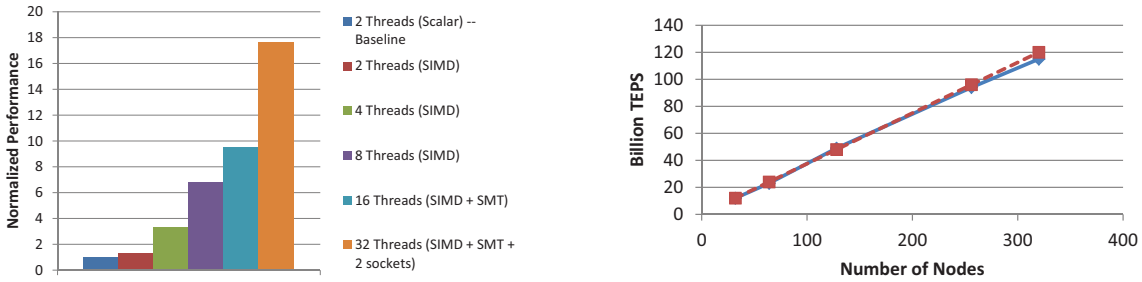
Platform Description: We evaluate our graph traversal algorithm on a 320-node Intel® Endeavor cluster. Each node has 64 GB RAM and is a dual-socket Intel® Xeon® E5-2670 with a total of 16 cores running at 2.6 GHz. The total number of cores is 5120. Each core has a 64 KB L1 cache and a 256 KB L2 cache. All cores within a socket share a last level L3 cache (LLC) of 20 MB. All caches are kept coherent. The different nodes are connected with a QDR Infiniband 4X with peak bandwidth of 4 GB/s and two-way 14-ary fat tree topology [30]. We also run our code on 1024 Westmere nodes and also 1024 Sandybridge nodes of the NASA Pleiadas supercomputer [31]. We use the Intel® Composer XE 2011 compiler and the Intel® MPI [25] 4.0.3 library in this work.

Energy consumption is a key metric that impacts the utility bills of modern data centers. The energy consumed during an application run is the product of the average power consumption during the run and execution time. In order to measure the power consumption during Graph runs, we use Intel® NodeManager, a part of Intel Data Center Manger SDK [19]. Intel® NodeManager works with power measurement counters in recent Intel® Xeon® E3, E5 and E7 processors, and performs real-time monitoring of power and inlet temperature data of clusters aggregated to rack, row, and room.

The Thermal Design Power (TDP) of the Intel® Xeon® E5-2670 is 115 Watts/socket or 230 Watts total. However, our power measurements yield numbers in the range of 240-300 Watts, including all additional sources of power consumption including memory, storage and cooling.

Data sets: We run on varying graph sizes in accordance with the Graph500 specification, with $\mathcal{V}$ varying from 2^{29} for 32-node runs to 2^{33} for 320-node runs. We used $\mathcal{V} = 2^{34}$ for our 1024 node Westmere and Sandybridge runs.

Scalability of our algorithm: Figure 5(a) presents the core scaling of our algorithm on 320 nodes relative to using 2 threads with scalar operations as baseline. We only present results for 2 threads onwards because one thread is reserved for performing MPI operations. The figure shows that we scale near-linearly from 2 to 8 threads (6.8X out of a peak of 7X using 7 compute threads). This code is limited by latency of



(a) Thread Scalability (b) Node Scalability

Fig. 5: **Scaling of graph traversal on Graph500 inputs with (a) number of threads and (b) number of nodes on a 320-node Sandybridge system. Thread scalability results include effects of SIMD and SMT, and are shown on 320 nodes. We obtain 1.2X SIMD benefits and overall thread scaling of 17.6X on 32 threads. Node scalability results are for weak scaling shown from 32 to 320 nodes. Due to our parallelization schemes, performance increase in super-linear for less than 32 nodes. We see near-linear scaling from 32 to 320 nodes.**

memory accesses. Once we use SMT (the 16-thread bar in the figure), this latency is hidden and performance improves by 1.4X. The code becomes bound by memory bandwidth, and therefore we do not get further scaling. Finally, we get socket scaling of about 1.85X. Our single node code is NUMA aware, leading to **low inter-socket communication**.

Figure 5(b) shows the weak scaling of our algorithm as we move from 32 to 320 nodes with $\mathcal{V}$ varying from 2^{29} to 2^{33}. The reason we start with 32 nodes is because of our threading model: at Steps 2, 5 and 6 of the algorithm, we divide computations for each destination node among threads (refer to Section IV for details). Thus we obtain super-linear scaling from 1 to 32 nodes, since thread scaling is also exposed along with node scaling – we do not present these results to get an accurate picture of node scaling. From 32 to 320 nodes, **we obtain near linear scaling** of 9.6X with peak of 10X. Note that our per-node performance on 320-nodes is **only 1.8X** off the optimized single node performance; this is due to the overheads of compression and decompression.

Performance results: Our overall performance on 320 nodes is about 115 BTEPS, which **matches the #2 result on the Nov 2011 Graph500 list** [2] with 5.6X fewer nodes. We also run our algorithm on 1024 Westmere nodes of the NASA Pleiadas supercomputer [31], and obtain 195 BTEPS. On 1024 Sandybridge nodes, we obtain 270 BTEPS.

Performance breakdown: Figure 6 shows relative performance and impact of various optimizations. All performance breakdowns in Figure 6 incorporate the reverse scheme. The first bar shows our optimized single-node implementation combined with Scheme 1 of sending 8 bytes per edge. In this implementation, the effect of the reverse algorithm is about 1.7X. This impact gradually decreases as we spend less time in the forward phase using our MPI optimizations; and the time spent in the reverse phase almost remains constant. This code spends about 58% of the time outside computation and communication; this low efficiency is due to the large impact of dynamic load imbalance among the nodes, with some nodes traversing more edges and transferring more data than others.

The second bar shows Scheme 2, where we perform de-duplication and transfer about 6 bytes/edge at the expense of

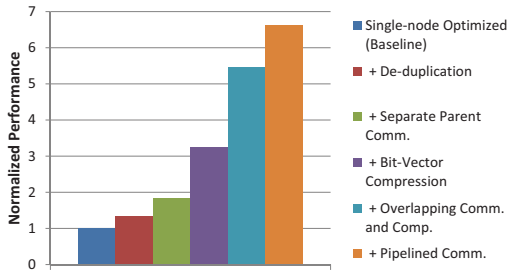


Fig. 6: **Performance comparison of our multi-node optimizations including compression schemes: namely data de-duplication (Scheme 2), separate parent communication , and bit-vector compression (Scheme 4); and latency hiding schemes: overlap of computation and communication and pipelined communication. Performance is normalized to the baseline of sending 4-bytes for vertex and parent id's (Scheme 1). The most optimized single node code is used for all comparisons. We obtain 3.2X better performance using compression schemes and further 2.1X using latency hiding techniques.**

performing about 1.6X more computation. Our performance is about 1.33X better than the previous scheme – roughly in ratio of overall bandwidth in spite of the increased computation. This is because our efficiency is slightly higher at 52%. We noted that the load-imbalanced nodes that sent more data and were responsible for the low efficiency also had more duplications; removing these leads to better load balance.

Our third bar shows separate parent communication along with Scheme 2. Our performance increases further by about 1.4X. There is a small additional computation cost due to the parent lookups in Step 5.

We then show Scheme 4, using bit-vector compression techniques. This significantly decreases MPI bandwidth requirements from about 4.13 bytes/edge to 2.63 bytes/edge, or about 1.6X improvement. For this scheme, we noted that the reverse algorithm only gives a 1.4X improvement in performance – since the forward phase has improved significantly in performance due to our algorithmic changes. The overall performance improvement of 1.75X from the previous scheme is larger than the MPI bandwidth decrease. The main reason is that the load imbalance reduces to about 20-30% of overall runtime; the bit-vectors sent are of constant length irrespective of load imbalance in the size of the $\mathcal{PBV}$ arrays sent. Our **overall gains from compression is about 3.2X**.

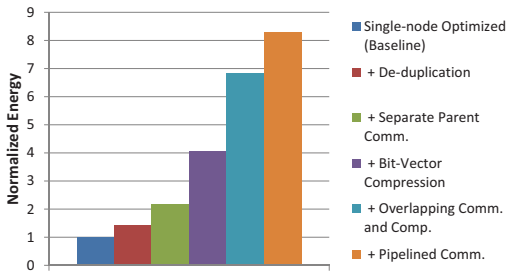


Fig. 7: **Comparison of energy efficiency of our various optimization steps. The graph shows energy efficiency normalized to the baseline of sending 4 bytes for vertex and parent id's (Scheme 1). We obtain an overall energy efficiency improvement of about 8.1X for our most optimized code. This efficiency comes due to improvements in performance (shown in Figure 6 as well as lower power consumption for optimized code due to lower MPI bandwidth requirements. Our node-level power measurements only capture part of the improvement; specifically the reduction in memory bandwidth to copy packet data.**

Finally, our two improvements of overlapping computation with communication and pipelining communication steps improves performance by 2.1X over Scheme 4 – giving **our final implementation an overall performance gain from Scheme 1 of about 6.6X**. The overlap of computation and communication gives about 1.7X in performance, while our pipelining optimizations results in further 1.2X improvement. In our final implementation, the time spent in computation and communication is comparable; and the overall time spent is only 5% off computation and communication costs.

Energy comparisons: Figure 7 shows our energy efficiency advantage from our optimized compression and latency hiding techniques. This advantage comes from both performance advantages shown in Figure 6 and additional power advantages due to lower inter-node communication by compression. While we do not directly measure the power used by the interconnect (we only measure per-compute node power), higher inter-node communication also results in higher memory bandwidth consumption on compute nodes due to copies of packets to and from user visible space. In practice, we see about 1.25X power savings (average of 240 Watts rather than 300 Watts for unoptimized code) from our optimizations. In addition to our performance advantages of 6.6X of optimized code, we obtain **overall energy efficiency improvements of about 8.1X**. We obtain overall energy efficiency of 1.5 MTEPS/Watt.

VI. Related Work

There have been many attempts to improve the speed of graph traversal on single-node systems including CPUs [29, 32–34] as well as other platforms [35–37]. Recently, Chhugani et al. [1] published an optimized single node CPU implementation of graph traversal that efficiency uses available memory bandwidth. We extend this implementation to multiple nodes.

However, large-scale graph analysis often requires processing terabytes of data [6] and large cluster machines are essential to even hold the data. The closest related work to this paper are the studies on large distributed graph processing on commodity CPU-based clusters [17, 18, 38]. They all ran the Graph500 benchmark for their analysis. Suzumara et al., [17] and Buluç et al. [18] show results on clusters of a few thousand nodes on commodity Intel Westmere EP and Cray XE6 consisting of AMD Magny-Cours processors respectively. Buluç et al. [18] report a performance of 17.8 GigaTEPS on a 40,000 core AMD Magny-Cours system for a scale=32 graph. In contrast, in this paper, we achieve 115 GigaTEPS (6.4X higher performance) using 5120 Intel Sandybridge cores (7.8X smaller cores). This is a combination of our algorithmic techniques as well as differences in underlying architecture. They improved their performance to 113.4 GigaTeps using an additional 8% cores. We match their performance (#2 on the Nov 2011 Graph500 list) using 5.6X fewer nodes. Suzumura et al. [17] report a performance of 11 GigaTEPS on 3072 Intel Westmere cores. We achieve 6.2X higher per-core performance (a small part of this is due to a different architectural generation).

There has been significant previous work in overlapping computation and communication to hide the latency of inter-node communications in clusters [39–41]. We adopt the technique described in these works and use a hybrid threading scheme with asynchronous data calls to perform the overlap. Our additional latency hiding technique of pipelining data transfers is based on the traditional software pipelining of instructions to increase instruction-level parallelism in optimizing compilers [42]. Here we adapt the technique to use it to overlap the communication of the (dependent) parent transfers with the boundary vertex transfers.

Energy efficiency is a key metric for Exascale challenge problems including graph traversal [7, 43]. To the best of our knowledge, this is the first paper showing energy efficiency on the Graph500 benchmark.

VII. Conclusion

In this paper, we showed that we can obtain high performance graph traversal on multi-node clusters, using a combination of efficient low-overhead data compression techniques to reduce transfer volumes along with latency-hiding techniques. Coupling our algorithmic novelty with an implementation that maximally exploits all micro-architectural dimensions of modern cluster-based multi-core processors, we achieved 6.6X performance improvements and almost an order-of-magnitude efficiency improvement for graph traversal. Our resulting Graph500 implementation was able to match the second ranked result in the most recent November 2011 Graph500 list with about 5.6X fewer nodes, with the highest per-node performance in the top 10 of the list.

We have two directions of future work. We are in the process of running our algorithm on larger clusters and are extending our compression schemes to include delta encoding and variable length compression. Second, we want to extend our power measurement methodology to account for the reduction in interconnect power consumption as we reduce communication.

A. Acknowledgments

We thank NASA for access to the Pleiades supercomputer, and Scott McMillan for obtaining our results on that system. We also thank Paresh Pattani for access to Intel Endeavor.
Generated by IEEEtran.bst, version: 1.12 (2007/01/11)

REFERENCES

[1] J. Chhugani, N. Satish, C. Kim, J. Sewall, and P. Dubey, "Fast and efficient graph traversal algorithm for cpus : Maximizing single-node efficiency," *Parallel and Distributed Processing Symposium, International*, vol. 0, pp. 1–10, 2012.

[2] "The Graph 500 List (Nov 2011)," http://www.graph500.org.

[3] H. Kwak, C. Lee, H. Park, and S. B. Moon, "What is twitter, a social network or a news media?" in *WWW*, 2010, pp. 591–600.

[4] A. Mislove, M. Marcon, P. K. Gummadi, P. Druschel, and B. Bhattacharjee, "Measurement and analysis of online social networks," in *Internet Measurement Conf.*, '07, pp. 29–42.

[5] M. Fernandez, D. Florescu, A. Levy, and D. Suciu, "A query language for a Web-site management system," *SIGMOD Rec.*, vol. 26, pp. 4–11, September 1997.

[6] U. Kang, C. E. Tsourakakis, A. P. Appel, C. Faloutsos, and J. Leskovec, "Radius plots for mining tera-byte scale graphs: Algorithms, patterns, and observations," in *SDM*, 2010, pp. 548–558.

[7] D. Bader, "Exascale Analytics of Massive Social Networks," 2010, AN09 Min-isymposium on High Performance Comp. on Massive Real-World Graphs.

[8] M. Anderson, "Better benchmarking for supercomputers," *IEEE Spectrum*, vol. 48, pp. 12–14, January 2011.

[9] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the graph 500," *Cray User's Group*, May 2010.

[10] P. Kogge and K. B. et al., "ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems," 2008.

[11] J. Dongarra, P.Beckman, et.al,, "The international exascale software project roadmap," *International Journal of High Performance Computing*, vol. 25, no. 1, pp. 3–60, 2011.

[12] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan, "Fawn: a fast array of wimpy nodes," in *SOSP*, 2009, pp. 1–14.

[13] J. G. Koomey, "Worldwide electricity used in data centers," *Environmental Research Letters*, vol. 3, no. 3, p. 034008, 2008. [Online]. Available: http://stacks.iop.org/1748-9326/3/i=3/a=034008

[14] Julian Seward, "Bzip2: Program and Library for lossless, block-sorting data compression. ," 2012. [Online]. Available: bzip.org

[15] K. M. Wong and S. Chen, "The entropy of ordered sequences and order statistics." *IEEE Transactions on Information Theory*, vol. 36, no. 2, pp. 276–284, 1990.

[16] H. E. Williams and J. Zobel, "Compressing integers for fast file access," *Comput. J.*, vol. 42, no. 3, pp. 193–201, 1999.

[17] Toyotaro Suzumura et al., "Performance characterization of the graph500 on large-scale distributed environment," in *IISWC*, 2011.

[18] A. Buluç and K. Madduri, "Parallel breadth-first search on distributed memory systems," in *SC*, 2011, p. 65.

[19] Intel, "Intel Data Center Manager," http://software.intel.com/sites/data-centermanager/, 2012.

[20] D. Bader, G. Cong, and J. Feo, "On the architectural requirements for efficient execution of graph algorithms," in *Parallel Processing, 2005. ICPP 2005. International Conference on*, june 2005, pp. 547 – 556.

[21] J. R. Crobak, J. W. Berry, K. Madduri, and D. A. Bader, "Advanced shortest paths algorithms on a massively-multithreaded architecture," in *IPDPS*, 2007, pp. 1–8.

[22] D. A. Bader, S. Kintali, K. Madduri, and M. Mihail, "Approximating betweenness centrality," in *WAW*, 2007, pp. 124–137.

[23] "Intel Advanced Vector Extensions Programming Reference," 2008, http://softwarecommunity.intel.com/isn/downloads/intelavx/Intel-AVX-Programming-Reference-31943302.pdf.

[24] "The Message Passing Interface (MPI) standard," http://www.mcs.anl.gov/research/projects/mpi/.

[25] "Intel MPI Library," http://software.intel.com/en-us/articles/intel-mpi-library/.

[26] T. Willhalm, N. Popovici, Y. Boshmaf, H. Plattner, A. Zeier, and J. Schaffner, "Simd-scan: ultra fast in-memory table scan using on-chip vector processing units," *Proc. VLDB Endow.*, vol. 2, no. 1, pp. 385–394, aug 2009.

[27] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey, "Fast: fast architecture sensitive tree search on modern cpus and gpus," in *SIGMOD*. ACM, 2010, pp. 339–350.

[28] R. Motwani and P. Raghvan, *Randomized Algorithms*. Cambridge University Press, 1995.

[29] S. Beamer, K. Asanovic, and D. A. Patterson, "Searching for a parent instead of fighting over children: A fast breadth-first search implementation for graph500," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2011-117, Nov 2011. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-117.html

[30] C. E. Leiserson, "Fat-Trees: Universal Networks for Hardware-Efficient Supercomputing," *IEEE Transactions on Computers*, vol. 34, no. 10, 1985.

[31] NASA, "Pleiadas supercomputer," http://www.nas.nasa.gov/hecc/resources/pleiades.html.

[32] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader, "Scalable graph exploration on multicore processors," in *SC*, '10, pp. 1–11.

[33] C. E. Leiserson and T. B. Schardl, "A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers)," in *SPAA*, 2010, pp. 303–314.

[34] Y. Xia and V. K. Prasanna, "Topologically Adaptive Parallel Breadth-first Search on Multicore-Processors," in *PDCS*, 2009.

[35] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, "Acc. CUDA graph algorithms at max. warp," in *PPOPP*, '11, pp. 267–276.

[36] D. P. Scarpazza, O. Villa, and F. Petrini, "Efficient breadth-first search on the cell/be processor," *IEEE Trans. Parallel Distrib. Syst.*, vol. 19, no. 10, pp. 1381–1395, 2008.

[37] A. Yoo, E. Chow, K. W. Henderson, W. M. III, B. Hendrickson, and Ü. V. Çatalyürek, "A scalable distributed parallel breadth-first search algorithm on bluegene/l," in *SC*, 2005, p. 25.

[38] J. Jose, S. Potluri, M. Luo *et al.*, "Upc queues for scalable graph traversals: Design and evaluation on infiniband clusters," in *PGAS*, 2011.

[39] V. Marjanović, J. Labarta, E. Ayguadé, and M. Valero, "Overlapping communication and computation by using a hybrid mpi/smpss approach," in *Proceedings of the 24th ACM International Conference on Supercomputing*, ser. ICS '10. New York, NY, USA: ACM, 2010, pp. 5–16. [Online]. Available: http://doi.acm.org/10.1145/1810085.1810091

[40] B. V. Protopopov and A. Skjellum, "A multithreaded message passing interface (mpi) architecture: performance and program issues," *J. Parallel Distrib. Comput.*, vol. 61, no. 4, pp. 449–466, Apr. 2001. [Online]. Available: http://dx.doi.org/10.1006/jpdc.2000.1674

[41] Mao Jiayin and Song Bo and Wu Yongwei and Yang Guangwen, "Overlapping 107 Communication and Computation in MPI by Multithreading," 2006.

[42] V. H. Allan, R. B. Jones, R. M. Lee, and S. J. Allan, "Software pipelining," *ACM Computing Surveys*, vol. 27, no. 3, pp. 367–432, 1995.

[43] H. Simon, "Exascale computing: Applications performance and energy efficiency," *Santa Barbara Summit on Energy Efficiency*, April 2011.